Distributed Envy Minimization for Resource Allocation*

Arnon Netzer and Amnon Meisels
Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, Israel

Keywords: Resource Allocation, Indivisible Goods, Envy Minimization, Distributed Search.

Abstract: The allocation of indivisible resources to multiple agents generates envy among the agents. An Envy Free allocation may not exist in general and one can search for a minimal envy allocation. The present paper proposes a formulation of this problem in a distributed search framework. Distributed Envy Minimization (DEM) - A Branch and Bound based distributed search algorithm for finding the envy minimizing allocation is presented and its correctness is proven. Two improvements to the DEM algorithm are presented - Forward Estimate (DEM-FE) and Forward Bound (DEM-FB). An experimental evaluation of the three algorithms demonstrates the benefit of using the Forward Estimate and Forward Bound techniques.

1 INTRODUCTION

Consider the allocation of resources (or tasks) to multiple agents, where agents associate their personal utilities to the allocated resources. A desirable allocation can in principle satisfy any of a number of social welfare functions.

In most cases the target social state is the Utilitarian state, widely known as *Social Welfare*, in which the goal is to maximize the sum of utilities of all agents (Rosenschein and Zlotkin, 1994; Moulin, 1988). However, in many cases reaching a *Fair* allocation may be more desirable than an *Efficient* one (Kleinberg et al., 2001; Lee et al., 2004). In some cases Fairness and Efficiency can be combined by looking for a Pareto Optimal Fair allocation (Chevaleyre et al., 2007). A key concept in the literature on *Fair Division* is *Envy Freeness* (Brams and Taylor, 1996). An allocation is envy free if no agent values another agent's bundle over its own.

A socially desirable allocation can be reached by multiple agents that use a negotiation framework (Endriss et al., 2006). However, such approaches typically require the existing of at least one divisible resource (money) in an adequate quantity. As a result, in the presence of money, reaching an Envy Free allocation can be addressed in a distributed negotiation framework (Asadpour and Saberi, 2007; Chevaleyre et al., 2007).

*The research was supported by the Lynn and William Frankel Center for Computer Science, and by the Paul Ivanier Center for Robotics and Production Management

In some cases the use of money may not be applicable. Consider the allocation of tasks to workers in a factory, or the allocation of shifts to nurses in a hospital ward. It is reasonable to assume that each nurse will have different preferences for shifts, and having nurses paying money to other nurses in order to switch shifts may be unacceptable. In this example we need all tasks to be allocated and an envy free allocation is clearly desirable. Unfortunately, when money is not involved, and all resources must be allocated, there is no guarantee that an envy free solution exists.

Consider the case of three agents and two resources in Figure 1. Denote by $u_i(r_j)$ the utility of agent i for getting resource j. It is easy see that in this example agent 1 is only interested in r_1, agent 3 is interested in r_2 , and agent 2 has a non zero utility for both resources. In fact, getting both resources is valuated by agent 2 more than the sum of the two single utilities. Since we have three agents and only two resource, at least one agent will end up getting nothing, and will necessarily be envious.

$u_1() = 0$	$u_2() = 0$	$u_3() = 0$
$u_1(r_1) = 3$	$u_2(r_1) = 3$	$u_3(r_1) = 0$
$u_1(r_2) = 0$	$u_2(r_2) = 6$	$u_3(r_2) = 4$
$u_1(r_1,r_2) = 3$	$u_2(r_1,r_2) = 10$	$u_3(r_1,r_2) = 4$

Figure 1: Example of utilities of three Agents, for two resources.

Maximizing Social Welfare in the present example, agent 2 would get both resources. For this allocation the sum of all utilities would be 10 (the utility of

Netzer A. and Meisels A..
Distributed Envy Minimization for Resource Allocation.
DOI: 10.5220/0004186700150024
In *Proceedings of the 5th International Conference on Agents and Artificial Intelligence* (ICAART-2013), pages 15-24
ISBN: 978-989-8565-38-9

agent 2) which is higher than the sum of utilities for any other allocation. However, in this allocation both agent 1 and 3 envy agent 2.

Since an envy free allocation may not exist, and even finding whether such an allocation exists is an NP-Complete problem (Bouveret and Lang, 2008), one can look for the allocation that minimizes the envy between the agents. The envy of any agent a_i of any other agent a_j may be measured in absolute terms - the utility that agent a_i associates with the bundle allocated to a_j minus the utility it associates with its own allocated bundle. Another option is to use a relative term - the utility a_i associates with the bundle allocated to a_j divided by the utility it associates with its own (Lipton et al., 2004).

Regardless of the method for computing the envy between two agents, there may be several global target functions for envy minimization. One may wish to minimize the number of envious agents, or the sum of all envy in the society (Utilitarian envy minimization). Alternatively, one may minimize the envy of the agent that is worst off (Egalitarian envy minimization), the agent with the largest amount of envy.

Recently, a centralized Branch and Bound algorithm for finding a fair allocation of indivisible goods was proposed in (Vetschera, 2010). In that work a centralized Branch and Bound algorithm was proposed for minimizing global target functions that represent fairness, such as Maxmin and Nash bargaining. Due to the nature of the problem, a distributed algorithm for finding an envy minimizing allocation is desirable.

The field of Distributed Constraint Reasoning provides a widely accepted framework for representing and solving Multi Agent Systems (MAS) problems. In a distributed constraint problem each agent holds a set of variables representing its state. These variables take values from a finite domain and are subject to constraints. A distributed constraint algorithm defines an interaction protocol for coordinating a joint assignment of variables.

Distributed Constraint Optimization Problems (DCOPs) were successfully applied to various MAS problems - coordinating mobile sensors (Lisý et al., 2010; Stranders et al., 2009), meeting and task scheduling (Maheswaran et al., 2004) and many others. Recent years have seen a large number of different algorithms for optimally solving DCOPs. These include Synchronous Branch and Bound (SBB) (Hirayama and Yokoo, 1997), BnB-ADOPT (Yeoh et al., 2010), ConcFb (Netzer et al., 2012) and others.

The present paper presents a formulation of envy minimization for indivisible goods allocation as a DCR problem. In this formulation an agent is constrained with another agent if both of them are "interested" in the same resource. For the example in Figure 1 this can be represented by the constraint graph in Figure 2. The variables of agents represent the resource that the agent is interested in and their allocation, and the interaction protocol defines the communication between agents in the constraint graph. So, a_2 is connected to a_1 since they are both interested in r_1, and to a_3 due to their common interest in r_2. a_1 and a_3 have no resource they are both interested in, and are not connected.

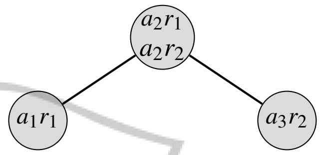


Figure 2: Constraints Graph for the example in Figure 1.

The formulation of envy minimization for indivisible resource allocation as a DCR problem enables the design of distributed algorithms for finding minimal envy solutions. The present paper presents a new Distributed Envy Minimization algorithm (DEM). Inspired by state of the art DCOP algorithms two improved algorithms are also presented (DEM-FE and DEM-FB) and the performance of the algorithms is compared.

The remainder of this paper is structured as follows: Section 2 formally defines envy minimization for indivisible resource allocation as a DCR problem. Section 3 presents the algorithms for solving these problems and the correctness and completeness proof of the algorithm are in Section 4. The experimental evaluation and the conclusions are in sections 5 and 6 respectively.

2 INDIVISIBLE RESOURCE ALLOCATION

2.1 Basic Definitions

An Indivisible Resource Allocation Problem consists of a set of agents $\mathcal{A} = \{a_1 \dots a_n\}$, and a finite set of indivisible resources $\mathcal{R} = \{r_1 \dots r_k\}$.

An agent allocation R_i is the set of resources allocated to agent a_i. An allocation $R_{\mathcal{A}}$ is a partitioning of $\mathcal{R}$ among the agents in $\mathcal{A}$. Formally: $R_{\mathcal{A}} = \{R_1 \dots R_n\}$ such that $R_i \cap R_j = \{\}$ for $i \neq j$ and $\bigcup_{i \in \mathcal{A}} R_i = \mathcal{R}$

In the general case every agent $a_i \in \mathcal{A}$ has a utility function u_i that maps an agent allocation K_i to a non negative utility ($u_i : 2^{\mathcal{R}} \rightarrow \mathbb{R}^+$). For the scope of this paper we will only consider super modular utility functions. So, for the scope of this paper $u_i(A \cup B) \geq u_i(A) + u_i(B) - u_i(A \cap B)$ for all $A, B \subseteq \mathcal{R}$. In order to avoid representation issues, our examples and pseudo code use additive utility functions.

An agent a_i envies another agent a_j if it valuates its allocation less than the allocation of the other agent: $u_i(R_i) < u_i(R_j)$ for $i, j \in \mathcal{A}$. Note that the envy of an agent depends only on the allocations and on that agent's utility function. The utility functions of the other agents are irrelevant for the envy of a given agent.

An allocation is Envy Free if every agent valuates its allocation at least as much as the allocation of any other agent. In other words, $R_{\mathcal{A}}$ is Envy Free iff $u_i(R_i) \geq u_i(R_j)$ for all $i, j \in \mathcal{A}$.

2.2 Envy Minimization

It is easy to see that an envy free allocation may not exist for Indivisible Resource Allocation. A simple example would be a system with two agents and one resource, that has a non zero utility for both agents. Since the resource can only be allocated to one of the agents, the other agent will envy. Since an Envy Free allocation requires that no agent envies any other agent, one may draw an analogy to constraint satisfaction problems in which no constraint can be violated.

When an Envy Free allocation does not exist, one may try to minimize the number of agents that are envious. This is analogous to MaxCSP (Larrosa and Meseguer, 1996) in which the goal is to minimize the number of violated constraints.

Returning to the example in Figure 1, allocating both r_1 and r_3 to agent 2 will maximize the social welfare, but leaves both agents a_1 and a_3 envious of agent a_2. A better allocation in this case may be to allocate r_1 to agent 1 and r_2 to agent 3. In this allocation only agent 2 is envious.

The amount of envy of agent i in agent j can be measured as $E_{ij} = u_i(R_j) - u_i(R_i)$, for all $i, j \in \mathcal{A}$ (where negative envy is truncated to 0). Another option is to measure relative envy: $E_{ij} = u_i(R_j)/u_i(R_i)$, for all $i, j \in \mathcal{A}$. When an agent envies more than one other agent, the agent's envy is taken to be its maximum envy of all other agents: $E_i = max_j(E_{ij})$.

Once the amount of envy of an agent is defined, one can set a global goal function for the envy of agents, and look for an allocation that minimizes this global function. This is analogues to a Constraint Optimization Problem. One example of such a global function would be the Utilitarian function, in which the goal is to minimize the sum of the envy of all agents. Another example may be the Egalitarian function, in which the goal is to minimize the envy of the "worst off" agent, the agent whose envy is the greatest.

If one uses the absolute envy between two agents in the example in Figure 1, minimizing the sum of all envies will result in the allocation of r_1 to agent a_1 and r_2 to agent a_2. This allocation will yield a total $envy = 4$ (only agent 3 is enviuos). Optimizing for the worst off agent will result in allocating r_1 to agent a_2 and r_2 to agent a_3. In this allocation the maximum envy of a single agent is 3 (for both a_1 and a_2) and this is the best allocation in terms of Egalitarian envy.

Figure 3 presents the search space for the utilities in Figure 1, for absolute envy and a global target of minimizing the sum of all envy. Each edge represents a variable, so, a_1r_1 is the variable that represents resource 1 allocated to agent a_1. The leafs are the global envy for the corresponding full allocation. An edge from a node down and right, represents a true assignment (the resource is allocated to this variable), in the same way an edge from a variable down and left, represents a false assignment (the resource is *not* allocated to this variable). The grayed out areas are illegal parts of the search space. A part of the search space is illegal either because it requires a resource to be allocated twice, or not to be allocated at all.

One can see that for this example there are only 4 legal full allocations and the optimal solution allocates r_1 to a_1 and r_2 to a_2, to get a global envy of 4. The only envious agent in this optimal allocation is agent a_3 which valuates the bundle of a_2 to be 4, and its own utility in the optimal allocation is 0.

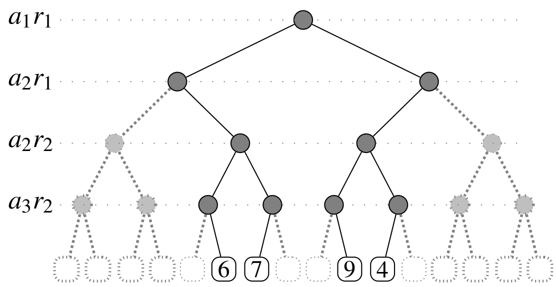


Figure 3: Serach space for the example in Figure 1.

3 DISTRIBUTED ENVY MINIMIZATION

Consider a distributed framework for a Branch and Bound search algorithm that finds the allocation of minimal Envy. This framework can be easily adapted for a variety of envy minimization target functions. The algorithms are based on Asymmetric Distributed Constraint Optimization Problems (ADCOP) (Grubshtein et al., 2009), with the required modification for envy minimization and for enforcing the global constraint that all resources must be allocated.

3.1 Algorithm Overview

In the proposed Distributed Envy Minimization (DEM) algorithm each agent a_i has a local boolean variable x_{ir} for each resource r for which a_i has a non zero utility $u_i(r) > 0$. Assigning *true* to x_{ir} means that r is allocated to agent a_i.

Each agent maintains a list of neighbors (*NB_List*) for each of its variables. The *NB_List* of a variable contains all other agents that are interested in the resource $NB\_List_{x_{ir}} = a_j : j \neq i, u_j(r) > 0$.

The search algorithm maintains an invariant attribute in which only one variable of all interested agents that represents resource r can be true. In addition, in a full allocation at least one of the variables that represents resource r must be true. This ensures that all resources are allocated, and that at no stage of the algorithm a resource is allocated to two agents.

All agents are ordered lexicographically. If agent a_i is before agent a_j in the lexicographic order, we say agent a_i is a higher priority agent than agent a_j (Meisels, 2007).

Each agent orders its variables in a lexicographic order. Each agent at its turn, tries to assign true to any variable which represents a resource that was not allocated by higher priority agents. Whenever an agent has all of its variables assigned ($true$ or $false$) it sends a message to the next agent in the global order, informing it on the assignments of all higher priority agents, and signaling it that it is its turn to assign variables.

Whenever an agent assigns true to a variable, it sends a message to all of the variable higher priority neighbors (agents in the variable *NB_List* that have higher priority than the current agent). Each such higher priority neighbor returns a message with its envy evaluation for the current agent. Based on the envy reports, and depending on the global minimization target function, the agent decides whether to keep the assignment or to backtrack.

If an agent needs to backtrack (change its assignment from true to false) on a variable that has no lower priority neighbors, it means that there is no other agent that can take this resource, and the agent needs to backtrack farther. If an agent needs to backtrack on a variable that is already assigned a false value, it needs to backtrack farther. If an agent needs to backtrack on its first variable, it backtracks to the previous agent.

Whenever the last agent successfully assigns all its variables, a new higher bound on the envy minimization target function has been found. If the first agent needs to backtrack on its first variable, then the search has ended, the upper bound on the envy minimization target function, is the minimal envy, and the full allocation that is associated with it, is the optimal allocation.

Consider the algorithm run example in Figure 4. The order of the agents is lexicographic. The first agent a_1 starts by assigning its variable r_1 to *true*. Next a_2 must assign its r_1 variable to false, since resource 1 was already allocated to agent a_1. Agent a_2 proceeds by assigning r_2 to true. Agent a_3 must assign its r_2 to false, and the upper bound of the global envy is calculated to be 4, which is the envy of agent a_3. Note that according to the definition of envy, agent a_2 is not envious of agent a_1 even though it has a non zero utility for r_1. The reason is that a_2 values its assigned bundle by 6, and values the bundle assign to a_1 (e.g. r_1) by 3, which is less. Agent a_3 then backtracks to agent a_2. If a_2 assigns false to its r_2 then its envy will be 9, which is higher than the upper bound, hence it backtracks on r_2. Since r_1 of agent 2 is already set to false, it backtracks on it too. Now agent a_1 changes its assignment of r_1 to false, followed by true assignments of a_2 to both its variables. At this stage agent a_3 is left with no resources to get, and the total envy is calculated to be 7 (3 for agent a_1 and 4 for agent a_3). Since this is more than the upper bound, agent a_3 backtracks without updating the upper bound. Again, agent a_2 knows that assigning false to its r_2 will breach the upper bound, and backtracks further until the algorithm terminates.

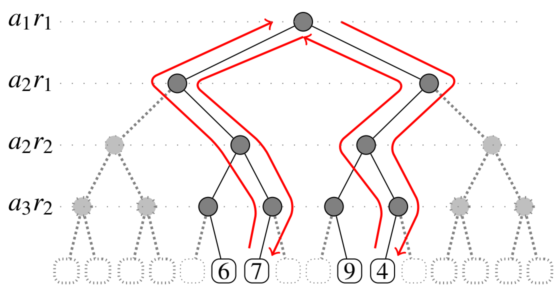


Figure 4: A Branch and Bound run on the search space in Figure 3.

3.2 DEM- Algorithm Description

The main data structures used by the algorithm are:

Agent_Assignment. A vector of boolean values representing the assignments of the agent's variables.

CPA. - A *CPA* (Current Partial Assignment) maintains all assignments of all variables of currently assigned agents. That is, it contains a set of pairs of the form $\langle Agent, Agent\_Assignment \rangle$

Envy_List. The *Envy_List* is a vector of Envy reported by all *assigned agents* with respect to a given CPA.

NB_List. A list of all agents that has a non zero utility as a given variable. The active SPs, held by each agent. the *NB_List* is maintained per variable per agent.

The algorithm uses four types of messages to transfer information and requests between agents:

CPA. A message containing a *CPA* and an *Envy_List*, sent by an agent after extending the CPA, to an unassigned agent.

BT_CPA. A backtrack message, notifying an agent that a CPA needs to be backtracked.

Envy_Request. A message containing a *CPA*, sent to an agent asking it to compute its envy for the given *CPA* and return it to the requesting agent.

Envy_Report. A message sent as a reply to **Envy_Request**, reporting the Envy for a given agent for a given *CPA*

The pseudo code of the main procedure of the DEM algorithm is described in Figure 5. It starts with the *initializing agent* calling Assign_Val() trying to assign its variables (line 3). The main loop (line 4) continuously looks for incoming messages (line 5), and dispatches them according to the message type to the appropriate functions (lines 7–15).

```
1  done ⟵ false
2  if Initializing_Agent then
3      Assign_Val(new CPA)
4  while not done do
5      msg ⟵ getNextMsg()
6      switch msg.type do
7          case CPA :
8              Receive_CPA(msg)
9          case BT_CPA :
10             Receive_BT_CPA(msg)
11         case Envy_Request :
12             Receive_Envy_Request(msg)
13         case Envy_Report :
14             Receive_Envy_Report(msg)
15         case Terminate :
16             done ⟵ true
```

Figure 5: main().

Figure 6 describes the pseudo code for Assign_Val() function. First, the function checks if all variables are assigned (line 1). If so, Agent_Assignment is completed and the appropriate function is called (line 2). Otherwise, the next unassigned variable is identified (line 4), and the *CPA* is checked to see if the resource represented by this variable is already assigned (line 5). If the resource is assigned then the variable gets a false value, the *CPA* is updated, and Assign_Val() is called again to try and assign the next variable (lines 7–9). If the resource was not assigned to a higher priority neighbor, then the variable is set to true, the *CPA* is updated (line 12). If the variable has no higher priority neighbors then its assignment cannot change the Envy valuation for any of the higher priority neighbors, and one can proceed to assign the next variable (lines 13–14). If there are higher priority neighbors an Envy_Request message is sent to them.

```
1  if all variables are assigned then
2      Agent_Assignment_Complete(CPA)
3  else
4      var ⟵ next unassigned variable
5      IsAssigned ⟵ check CPA if relevant good
       already assigned
6      if IsAssigned then
7          var = false
8          Update_CPA(CPA, var)
9          Assign_Val(CPA)
10     else
11         var = true
12         Update_CPA(CPA, var)
13         if var has no higher priority neighbors then
14             Assign_Val(CPA);
15         else
16             send Envy_Request to all higher priority
               neighbors
```

Figure 6: Assign_Val(*CPA*).

```
1  Envy ⟵ Calc_Envy(CPA)
2  Envy_List.put(agent, Envy)
3  Global_Envy ⟵ Calc_Global_Envy(Envy_List)
4  if Global_Envy ≥ Upper_Bound then
5      Backtrack(CPA)
6  else
7      if Last_Agent then
8          Upper_Bound ⟵ Global_Envy
9          Backtrack(CPA)
10     else
11         send CPA message to next agent
```

Figure 7: Agent_Assignment_Complete(*CPA*).

When an agent reaches a full Agent_Assignment (Figure 7), the agent calculates its envy against all higher priority agents (line 1). The global target function is then calculated based on the envy of the agent and all its higher priority agents (line 3). if the upper bound known for the global target function is breached, we do not need to proceed, and Backtrck()

is called (lines 4–5). Otherwise, if this is the last agent, a new upper bound is registered, and a BackTrak() is called (lines 7–9). If this is not the last agent then the CPA message is sent to the next agent (line 11).

Upon Backtrack() (Figure 8), if a backtrack is needed to a higher priority agent, then, if this is the first agent, the algorithm terminats (lines 3–5), and if not, a Backtrack message is sent. If the backtrack is to another variable in the agent then if the current variable is already assigned $false$, or if there is no lower priority agent that can take the relevant resource (line 9), there is no valid assignment for the variable and we need to backtrack further (lines 10–11). If the variable is assigned $true$ and there is some lower priority agent that can take the resource, the variable gets $false$ and we proceed to assign the next variable (lines 13–14).

```
1  var ⟸ last assigned variable
2  if var is first variable then
3      if Initializing_Agent then
4          Done ⟵ true
5          send terminate message to all agents
6      else
7          send Backtrack message to previous agent
8  else
9      if var == false or var has no lower priority
       neighbors then
10         remove var from CPA
11         Backtrack(CPA)
12     else
13         var = false
14         Assign_val(CPA)
```

Figure 8: Backtrack(*msg*).

In response to an Envy_Request message (Figure 9) the agent calculates its envy against the *CPA* in the message, and sends it back to the requesting agent (lines 2–3). When an Envy_Report message is received (Figure 10), the *Envy_List* is updated with the new envy. If the Envy_Reports of all higher priority agents were received, the global envy target function is calculated and compared to the known upper bound (lines 3–4). If the upper bound was breached a backtrack is issued, otherwise we proceed to assign the next variable.

```
1  CPA ⟵ msg.CPA
2  Envy ⟵ calc envy to CPA
3  send back Envy_Report message
```

Figure 9: Receive_Envy_Request(*msg*).

```
1  Envy_List.put(msg.sender, msg.envy)
2  if Envy_Report received from all higher priority
   neighbors then
3      Global_Envy ⟵ Calc_Global_Envy(Envy_List)
4      if Global_Envy ≥ Upper_Bound then
5          Backtrack(CPA)
6      else
7          Assign_Val(CPA)
```

Figure 10: Receive_Envy_Report(*msg*).

3.3 Forward Estimate - DEM-FE

Upon receiving an Envy_Request message from a lower priority neighbor, an agent a_i calculates its envy toward all agents on the CPA (Figure 9 line 2). However, there may be resources with positive utility to a_i which are not yet allocated to any agent on the CPA. Since eventually all resources will be allocated, if the utility of a_i for any of the resources not allocated on the CPA is larger than the bundle of any agent on the CPA, this can be used as a better bound on the envy of a_i. Note that since and does not know how resources not currently assigned on the *CPA* would be allocated, one can only consider the utility of each resource by itself, and not the utility of bundles of unallocated resources.

In order to incorporate the Forward Estimate (FE) capability, the only change needed is in the Received_Envy_Request() function. Figure 11 presents the enhanced function. Line 3 loops through all resources r_j for which agent a_i has a non zero utility , and are not yet allocated. For each of them, if the utility of agent $a_i(r_j)$ is higher than the calculated envy (line 4), the envy is updated accordingly (line 5).

```
1  CPA ⟵ msg.CPA
2  Envy ⟵ calc envy to CPA
3  foreach r_j not in CPA do
4      if u_i(r_j) > Envy then
5          Envy ⟵ u_i(r_j)
6  send back Envy_Report message
```

Figure 11: Receive_Envy_Request(*msg*)-FE.

3.4 Forward Bounding - DEM-FB

Forward Bounding is a method in which agents send the CPA to lower priority, unassigned agents, and receive bounds on what the valuation of these lower priority agents may be if the CPA will be extended to the responding agents. Though this method increases the computation and communication needed for assigning a new value, it may lead to a better pruning of the search space. Forward bounding have been shown to give a significant boost in DCOP algorithms (Gershman et al., 2009). In this section we show how forward bounding can be added to the distributed envy minimization algorithm described above.

The adaptation that is required is in the function Assign_Val(). Here we need to send Envy_Request (Figure 6 line 16) to all neighbors and not only to higher priority ones. In the same way in function Recieve_Envy_Report (Figure 10 line 2), the condition needs to be modified to wait for Envy_Reports from all neighbors.

The last modification needed is in the envy computation done by lower priority agents receiving an Envy_Request message. Since a lower priority agent receiving an Envy_Request does not have its variables assigned yet, it can only give a bound on its envy. The highest evaluating bundle that such an agent may be allocated by extending the current CPA would be all resources not already allocated on the CPA. So the agent computes its envy based on the assumption that its allocation would be composed of not yet allocated resources.

The new Receive_Envy_Request() routine is described in Figure 12. In line 2 the agent checks if the Envy_Request was originated by a higher priority agent. If it was (line 3), the agent assumes its assignment is all the resources currently unassigned on the CPA. The Envy is computed (line 4) based on either the agent assignment on the CPA (in case of a higher priority agent) or on the tentative assignment (in case of a lower priority one).

1 $CPA \longleftarrow msg.CPA$
2 **if** *msg.sender is higher priority agent* **then**
3     $My\_Tentative\_Assignment \longleftarrow$ all unassigned resources
4 $Envy \longleftarrow$ calc envy to CPA
5 **foreach** r_j *not in CPA* **do**
6   **if** $u_i(r_j) > Envy$ **then**
7         $Envy \longleftarrow u_i(r_j)$
8 send back Envy_Report message

Figure 12: Receive_Envy_Request(*msg*)-FB.

3.5 Envy Target Functions

The algorithms in sections 3.2, 3.3 and 3.4 can support all target functions of section 2.2. Supporting absolute envy measures or relative ones will require the correct envy calculation in Receive_Envy_Request() (Figure 9, line 2), and similarly Agent_Assignment_Complete() (Figure 7, line 1).

A Utilitarian envy minimization is achieved by setting the global envy calculation in Receive_Envy_Report() (Figure 10, line 3) and in Agent_Assignment_Complete() (Figure 7, line 3) to be the sum of the envy of all agents. An Egalitarian global envy will require setting the same global envy calculation to be the maximum of the minimal envy among all agents.

In order to minimize the number of agents with non zero envy, one can use a global envy calculation that adds 1 for every agent that has a non zero individual envy. Requiring an Envy Free solution is identical to minimizing the number of envious agents with the $Upper\_Bound$ set to 1.

4 ALGORITHM CORRECTNESS

To prove the algorithm's correctness, one first proves that it terminates and then proves that upon termination the value of the upper bound is the optimal envy (completeness).

To prove that the algorithm terminates one needs to prove that it will never go into an endless loop. To do so, one needs to consider the algorithm's state. A state $s_i \in S_i$ includes all agents, their variables and current assignment. The following Lemma proves that the same state is not generated more than once.

Lemma 1 (Unique States)**.** *A state S is never repeated.*

Proof. Assume by negation that some partial assignment $\{\langle a_1, v_1\rangle \dots \langle a_l, v_k\rangle\} = S_{lk}$ has bean duplicated. There is some agent $a_i (1 \leq i \leq l)$ who is holding the CPA and by assigning $\langle a_i, v_j\rangle$ on it, generates *for the first time* the duplicate partial assignment. Clearly, being the first duplication of the CPA means that a_i is the highest in the order of agents to assign itself the same assignment for the second time, with the same partial assignment before it.

Any new assignment added to the CPA is selected in the Assign_Val function. This function is invoked from either one of the following functions:

- main() - This function only invokes Assign_Val once - at the beginning of the run. Hence it cannot cause the same state to be produced more than once.
- Receive_CPA() - The Receive_CPA() function is invoked whenever a higher priority agent a_j (where $j < i$) sends a CPA message to a_i (line 8 of main()). A duplicated CPA generated by a_i includes the same assignments to all of its variables and therefore the first j assignments must be the same. This contradicts the assumption that a_i is the first agent which repeats a state.

- Backtrack() - If Assign_CPA() is invoked following line 14 of Backtrack(), line 13 was also executed. Specifically, a variable that had a $true$ value, is now set to $false$. As a result, Assign_Val() can never generate a duplicate CPA, which contradicts our assumption.

□

Theorem 1 (Termination)**.** *Every run of the algorithm terminates.*

Proof. The algorithm will terminate if the following conditions hold:

- The number of states it goes through is finite.
- It does not examine the same state more than once.
- The algorithm maintains progress. That is, it moves from one state to another within a finite amount of time.

The first condition is trivially met by the fact that the number of agents and the number of resources are finite. The second one immediately follows from Lemma 1.

Consider the state $s_a \in S_i$. This state can proceed to some other state $s_b \in S_i$ whenever the Assign_CPA() and Receive_BT_CPA() functions are executed (assigning $true$ or $false$ to a variable) by some agent. The only situation in which the algorithm does not move trivially to the next state is when the algorithm asks for Envy valuation of its neighbors following an assignment (Assign_val() line 16). In this case Envy_Request will be sent to all neighbors and the agent will wait for new messages to arrive. However, since every agent receiving an Envy_Request responds to it by an Envy_Report (Receive_Envy_Request() line 3), the agent assigning the new value is guarantied to receive Envy_Reports messages from every neighbor. This will result in either Backtrack() or a new Assign_Val() call in Receive_Envy_Report() lines 5 an 7 respectively.

□

To prove completeness one needs to prove that the value returned upon completion is indeed the optimal envy for a full allocation. We start by proving a monotonicity characteristic of the CPA.

Lemma 2 (CPA Monotonicity)**.** *Any extension of a CPA whose current envy is higher than a given upper bound will also be higher than that upper bound.*

Proof. The proof is divided into two parts. First one needs to show that the envy between two agents cannot decrease when the CPA is extended. Then, one shows that for the set of global target functions, global envy cannot decrease unless some agent's envy decreases.

The envy between two agents can be measured between a fully assigned agent and any other agent (Agent_Assign_Complete() line 1 or Receive_Envy_Request() line 2). Alternatively, envy can be measured between an unassigned agent and any other agent (in case of forward bound Receive_Envy_Request() line 4). For the first option, due to the fact that utilities are super-modular, and since a CPA extension can only add resources to agents that were not fully assigned, the envy of them cannot decrease. The second option deals with the forward bounding mechanism which assumes that agents will be allocated all available resources (see 12 line 3). Any extension of the CPA cannot result in the future agent getting more resources than was assumed, and due to super-modularity cannot result in a higher utility, which means that its envy can only increase.

For the scope of this paper three global target functions are considered: 1) the number of envious agents 2) the sum of envy of all agents 3) the envy of the agent that has the highest envy (see section 2.2). It is easy to see that for any of these target functions for the global envy to decrease, the envy of at least one agent must decrease.

□

We now prove the completeness of the algorithm.

Proof. Upon termination the result is the upper bound and the allocation that produced this upper bound. One needs to prove that the last reported upper bound is the minimal envy. Every full allocation envy is compared to the known upper bound (in Agent_Assignment_Complete line 4), and if it is lower, the upper bound is replaced by the new value (same place line 8). One needs to show that every allocation that will improve the upper bound will be checked.

If a full allocation is not checked, then it must have been pruned in the search process by backtracking on one of its possible partial assignments. For this not to violate completeness two conditions need to hold:

- Every CPA not extended has a higher global envy valuation than the upper bound.
- Every potential extension of a CPA not extended will have a higher global envy valuation than the upper bound.

For the first condition to hold we observe that a CPA is not extended only if a Backtrack() was called for the given CPA. A Backtrack() is called from the following locations:

- Agent_Assignment_Complete() line 9 - called only after the CPA envy is checked against the upper bound (lines 1–4).

- Receive_Envy_Report() line 5 - conditioned on the CPA envy exceeding the upper bound (line 4).
- Backtrack() line 11 - this recursive call for Backtrack() is conditioned on the fact that the CPA cannot be extended. Either because the relevant variable is already assigned false, or that if the relevant variable will not get the resource allocated to it, no other variable can get it (line 9).

The second condition follows immediately from Lemma 1 and 2.

□

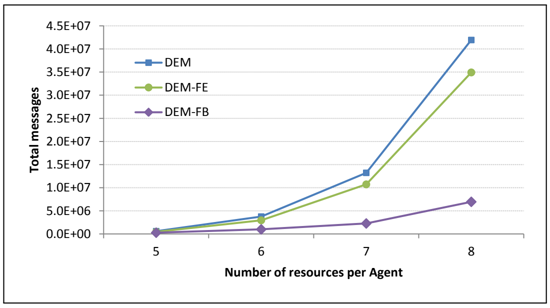
5 EXPERIMENTAL EVALUATION

Two performance measures are routinely used to evaluate distributed search algorithms: network load measured by the total number of messages sent (Lynch, 1996; Yokoo, 2000) and run-time in the form of Non-Concurrent Logic Operations (NCLOs) (Zivan and Meisels, 2006). In DCOPs the measure of NCLO usually translates to Non-Concurrent Constraint Checks. For envy minimization the logic operation is taken to be the evaluation of utility of a bundle of resources.

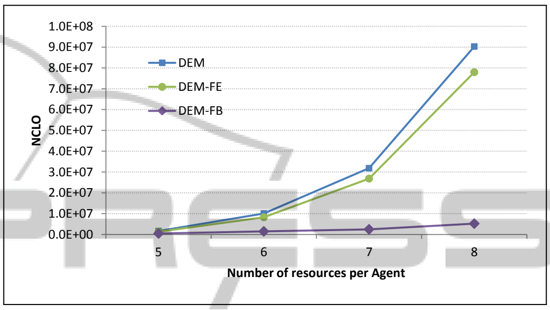
The first experimental setup included 10 agents, 15 resources and the number of resources per agent was varied between 5 and 8. The utility functions where additive and each agent randomly assigned a value in the range of 1–100, to each resource it was interested in. Each agent was randomly assigned resources it was interested in. The envy between two agents was taken to be the absolute envy, and the global optimization goal was the egalitarian social welfare function.

Figure 13 shows a comparison of DEM, DEM-FE and DEM-FB. Each point in the graph represents the average result for 50 randomly generated problems. The graph clearly demonstrates the pruning power of forward bounding, resulting in better performance of DEM-FB in both total message count and NCLO time.

The second experiment (Figure 14) included 10 agents and 15 resources, 5 resources per agent and the number of agents was varied from 10 to 14. As before, utility functions where additive and each agent randomly assigned a value in the range of 1–100 to each resource it was interested in. One can see that the performance enhancements between DEM and DEM-FE and between DEM-FE and DEM-FB, resemble the enhancements observed in the first experiment.

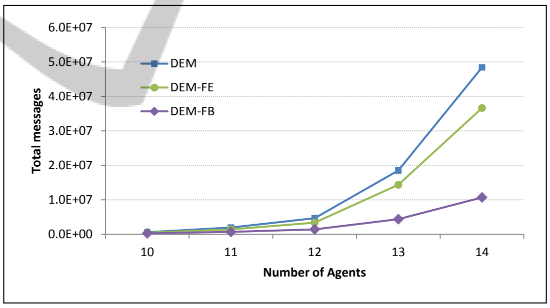


(a)

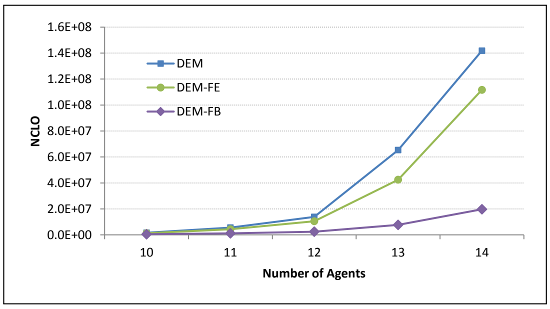


(b)

Figure 13: Algorithms comparison, 10 agents, 15 resources, number of resources per agent varies from 5 to 8.



(c)



(d)

Figure 14: Number of agents varied from 10 to 14.

6 CONCLUSIONS

The problem of envy minimization among agents that are assigned indivisible resources has been formulated and a distributed algorithm for finding an allocation that minimizes envy among agents has been proposed. Envy minimization generalizes the envy freeness idea, which does not exist in general for the allocation of indivisible resources.

The Distributed Envy Minimization (DEM) algorithm has been proven correct, and two extensions presented. One uses Forward Estimation to bound the amount of potential envy by unassigned agents (DEM-FB). The other extension bounds envy by considering potential allocation to unassigned agents (DEM-FE).

Several global target functions are described, from minimizing the sum of the total envy of all agents, to the amount of envy of the most envious agent (the Egalitarian version of envy minimization).

All algorithms have been evaluated empirically on randomly generated distributed envy minimization problems. The DEM-FB algorithm performs best on the random resource allocation problems that were generated, in both performance measures: non-concurrent run-time and network load. The results hold consistently both for a range of number of agents and for a range of number of resources.

REFERENCES

Asadpour, A. and Saberi, A. (2007). An approximation algorithm for max-min fair allocation of indivisible goods. In *STOC*, pages 114–121.

Bouveret, S. and Lang, J. (2008). Efficiency and envy-freeness in fair division of indivisible goods: Logical representation and complexity. *J. Artif. Intell. Res. (JAIR)*, 32:525–564.

Brams, S. J. and Taylor, A. D. (1996). *Fair division - from cake-cutting to dispute resolution*. Cambridge University Press.

Chevaleyre, Y., Endriss, U., Estivie, S., and Maudet, N. (2007). Reaching envy-free states in distributed negotiation settings. In *IJCAI*, pages 1239–1244.

Endriss, U., Maudet, N., Sadri, F., and Toni, F. (2006). Negotiating socially optimal allocations of resources. *J. Artif. Intell. Res. (JAIR)*, 25:315–348.

Gershman, A., Meisels, A., and Zivan, R. (2009). Asynchronous forward bounding. *Journal of Artificial Intelligence Research*, 34:25–46.

Grubshtein, A., Grinshpoun, T., Meisels, A., and Zivan, R. (2009). Asymmetric distributed constraint optimization. In *IJCAI-09*, Pasadena.

Hirayama, K. and Yokoo, M. (1997). Distributed partial constraint satisfaction problem. In *Proc. 3rd Intern. Conf. Princ. Pract. Const. Prog. (CP-97)*, pages 222–236.

Kleinberg, J. M., Rabani, Y., and Tardos, É. (2001). Fairness in routing and load balancing. *J. Comput. Syst. Sci.*, 63(1):2–20.

Larrosa, J. and Meseguer, P. (1996). Phase transition in max-csp. In *Proc. 12th European Conference on Artificial Intelligence (ECAI-96)*, pages 190–194, Budapest, Hungary.

Lee, C. Y., Moon, Y. P., and Cho, Y. J. (2004). A lexicographically fair allocation of discrete bandwidth for multirate multicast traffics. *Computers & OR*, 31(14):2349–2363.

Lipton, R. J., Markakis, E., Mossel, E., and Saberi, A. (2004). On approximately fair allocations of indivisible goods. In *ACM Conference on Electronic Commerce*, pages 125–131.

Lisý, V., Zivan, R., Sycara, K. P., and Pechoucek, M. (2010). Deception in networks of mobile sensing agents. In *9th Intern. Conf. Auton. Agents Mult. Sys. (AAMAS-10)*, pages 1031–1038, Toronto, Canada.

Lynch, N. A. (1996). *Distributed Algorithms*. Morgan Kaufmann.

Maheswaran, R. T., Tambe, M., Bowring, E., Pearce, J. P., and Varakantham, P. (2004). Taking DCOP to the real world: Efficient complete solutions for distributed multi-event scheduling. In *3rd Intern. Joint Conf. Auton. Agents Mult. Sys. (AAMAS-04)*, pages 310–317, New York, USA.

Meisels, A. (2007). *Distributed Search by Constrained Agents: Algorithms, Performance, Communication*. Springer Verlag.

Moulin, H. (1988). *Axioms of Cooperative Decision Making*. Cambridge University Press.

Netzer, A., Grubshtein, A., and Meisels, A. (2012). Concurrent forward bounding for distributed constraint optimization problems. *Artificial Intelligence*, 193(0):186 – 216.

Rosenschein, J. S. and Zlotkin, G. (1994). *Rules of Encounter - Designing Conventions for Automated Negotiation among Computers*. MIT Press.

Stranders, R., Farinelli, A., Rogers, A., and Jennings, N. R. (2009). Decentralised coordination of continuously valued control parameters using the max-sum algorithm. In *Proc. 8th Intern. Joint Conf. Auton. Agents Mult. Sys. (AAMAS-09)*, pages 601–608, Budapest, Hungary.

Vetschera, R. (2010). A general branch-and-bound algorithm for fair division problems. *Computers & OR*, 37(12):2121–2130.

Yeoh, W., Felner, A., and Koenig, S. (2010). BnB-ADOPT: An asynchronous branch-and-bound DCOP algorithm. *Jour. Artif. Intell. Res. (JAIR)*, 38:85–133.

Yokoo, M. (2000). Algorithms for distributed constraint satisfaction problems: A review. *Autonomous Agents and Multi-Agent Systems*, 3:198–212.

Zivan, R. and Meisels, A. (2006). Message delay and DisCSP search algorithms. *Annals of Mathematics and Artificial Intelligence (AMAI)*, 46:415–439.